Summary Prefix Tree: An over DHT Indexing Data Structure for Efficient Superset Search

Bassirou Ngom, Mesaac Makpangou

Summary Prefix Tree: An over DHT Indexing Data Structure for Efficient Superset Search

Bassirou Ngom*†, Mesaac Makpangou*
* Sorbonne Universités, UPMC Univ Paris 06, CNRS
INRIA - LIP6, Paris, France
Email: [bassirou.ngom, mesaac.makpangou]@lip6.fr
† Université Cheikh Anta Diop
Faculté des sciences et techniques – Département Mathématiques-Informatique
Dakar - Sénégal
Email: bassirou83.ngom@ucad.edu.sn

***Abstract*—This paper presents the summary prefix tree (SPT), a trie data structure that supports efficient superset searches over DHT. Each document is summarized by a Bloom filter which is then used by SPT to index this document. SPT implements an hybrid lookup procedure that is well-adapted to sparse indexing keys such as Bloom filters. It also proposes a mapping function that permits to mitigate the impact of the skewness of SPT due to the sparsity of Bloom filters, especially when they contain only few words. To perform efficient superset searches, SPT maintains on each node a local view of the global tree. The main contributions are the following. First, the approximation of the superset relationship among keyword-sets by the descendance relationship among Bloom filters. Second, the use of a summary prefix tree (SPT), a trie indexing data structure, for keyword-based search over DHT. Third, an hybrid lookup procedure which exploits the sparsity of Bloom filters to offer good performances. Finally, an algorithm that exploits SPT to efficiently find descriptions that are supersets of query keywords.**

I. Introduction

On the one hand, many applications can benefit from a scalable, fault-tolerant, and robust distributed keyword-based searching system to improve information sharing among peers. For instance in many developing countries, universities do not have enough resources (e.g., bandwidth, servers) to make their scientific publications widely available online. These universities also have limited resources dedicated to accesses to digital libraries. As a consequence, most researchers and students lack a common view of what research projects are undertaken on different universities, often leading to similar projects which ignore each other. Also, scarce resources are often wasted to download multiple times documents that are available nearby. A nation-wide index of documents available on peers, offering efficient keyword-based searches, can help improve the accessibility of scientific information to members of the community with no extra cost.

On the other hand, Distributed Hash Table (DHT) is a widely used building block for scalable, fault-tolerant, and robust peer-to-peer systems. However, supporting efficient scalable keyword-based search over a DHT is a challenge. Several proposals [1], [2], [3], [4], [5] that rely on distributed Inverted indices are confronted to a number of drawbacks, mainly high bandwidth consumption, uneven load of nodes, and the weak filtering of information when querying with popular keywords. Joung et al [6] propose to rely on a r-dimension hypercube to build a distributed index for keyword-based search over structured P2P networks. While this proposal is interesting especially for exact search, it exhibits poor performances for superset search. While a number of proposals [7], [8], [9], [10] rely on trie data structures to efficiently support complex queries over DHT, this approach is insufficiently investigated for keyword-based search.

This paper extends the use of a trie data structure to build an index supporting keyword-based search. It presents four main contributions. First, we approximate the superset relationship among keyword-sets by the descendance relationship among Bloom filters. Such a transformation permits to deal with compact set summaries, together with efficient bitwise operations. Second, we define a summary prefix tree, a trie data structure used to build a scalable, fault-tolerant and robust over DHT index for keyword-based search. Third, we define an hybrid lookup procedure that suits the specific of sparse indexing keys. Unlike the traditional linear lookup that tests all prefixes until it reaches the one that identifies the appropriate leaf node, the hybrid procedure jumps directly to prefixes that are significant for superset search. This procedure is particulary efficient for the location of sparse indexing keys. Finally, we propose a superset search algorithm that exploits the proposed SPT trie data structure to efficiently retrieve satisfying documents.

The rest of the paper is organized as follows. Section II presents existing related work. Then section III introduces the use of Bloom filter to approximate superset tests. Section IV presents the summary prefix tree and its main operations, while section V discusses our superset search algorithm. Section VI concentrates on the performance evaluation. Finally, section VII draws some conclusions and points some perspectives.

II. Related Work

A number of over-DHT keyword-based searching systems [1], [2], [3], [4], [11] rely on Distributed Inverted Indices. Distributed Inverted Indices are confronted to a number of drawbacks: high bandwidth consumption, uneven load of nodes, and the weak filtering of information when querying with popular keywords.

To avoid the drawbacks inherent to Distributed Inverted Indices, Joung et al [6] propose to rely on a r-dimension hypercube to build a distributed index for keyword-based search over structured P2P networks. They propose to use a hash function to summarize each description by a r-bits vector. Each document is indexed by the server associated with the r-bits that characterizes its description. In that proposal, Bloom filters are used as a mean to group documents into separated clusters, while in our SPT proposal Bloom filters serve to approximate superset relationship. Hence SPT needs to build Bloom filters that guarantee a false positive rate lower than some predefined threshold. A second difference is that, rather than relying on an ad'hoc indexing data structure such as an hypercube, SPT is an over-DHT indexing scheme that relies on a prefix tree, a trie data structure.

Mkey [12] is an overlay dependant indexing system. Mkey derives from each description summary a set of node identifiers, then replicates this summary on the identified servers. To search items that match a query keyword set, Mkey determines the identifiers of servers that are responsibles of indexing items that match this request and send them the request. The result is obtained by making the union set of results returned by different indexing servers. While Mkey and SPT rely on Bloom filter to represent descriptions, SPT is an over-DHT while Mkey is an overlay dependant solution.

A number of over-DHT indexing schemes [7], [13], [8], [9] have been proposed to support complex queries over DHT. Amoung these, LIGHT[8] is the closest to our SPT proposal. LIGHT relies on three novel features namely: space partitioning tree, tree summarization, and naming function. Our SPT proposal is largely inspired by LIGHT [8]. In particular, we use a naming function to map SPT leaf nodes into the DHT. We also adopt the tree summarization strategy to maintain, at each node, a view of the global summary prefix tree.

III. Superset Test Approximation

Bloom filters are compact data structures used in several distinct contexts to approximate membership tests.

Let S_m designate the set of Bloom filters of lenght m bits constructed thanks to the same set of hash functions, summaries of keyword-sets. We define the relationship $descendant$ among the elements of S_m, noted $\hookrightarrow$, as follows: $\forall f, q \in S_m, f \hookrightarrow q \Leftrightarrow (q \wedge f) == q)$ where $\wedge$ is the bitwise intersection of bit strings.

Given two keyword-sets F_1 and F_2, we decide that F_1 contains F_2 if $(f_1 \hookrightarrow f_2)$ where f_1 and f_2 are the summaries of F_1 and F_2 respectively, of same lenght and constructed thanks to the same hash functions.

This decision is an approximation with a risk of a false positive decision (that is deciding that F_1 is a superset of F_2 while it is not true). If $nmax$ designates the maximum number of keywords in F_1 and F_2, the higher the ratio ($\frac{m}{nmax}$), the lower the risk for false positive decision for membership test approximations [14] and hence for the superset test. In this paper we assume that m is determined such as to ensure an acceptable false positive rate for superset tests.

Using the above superset test approximation, we tranform the superset search problem to the "summary descendant search problem". To address this new problem, we propose to index descriptions summaries and to offer efficient descendant search operations.

IV. Summary Prefix Tree

Summary Prefix Tree (SPT) is a trie data structure that indexes 2-tuples, (*summary*, *docURI*), where summary is the Bloom filter that summarizes the description associated with the document identified by *docURI*. SPT is a binary tree. Any SPT node is uniquely identified by its label constituated of the root label concatenated with the labels of branches from the root through that node. By convention, the SPT root node is labelled "/"; the left branch departing from any internal node is labelled 0 while the right one is labelled 1. Each leaf node is associated a bucket that contains this node's label, a store of the set of records under the responsability of this node, and this node status (i.e., LEAF).

To distributed nodes buckets over a DHT, we define the function, $skey()$, that maps each SPT node identifier to a DHT storage key. Equation 1 sketches the mapping algorithm: nid is the node identifier to map to the DHT, z (resp. p) designates the longest prefix of nid with its rightmost bit equal to 0 (resp. 1), and $[0]*$ (resp. $[1]*$) refers to a sequence of bit 0 (resp. 1) repeted zero or more times.

$$skey(nid) = \begin{cases} "/", & if\ nid = "/" \\ "/0", & if\ nid = "/0[0]*" \\ "/1", & if\ nid = "/1[1]*" \\ "/p0", & if\ nid = "/p0[0]*" \\ "/z1", & if\ nid = "/z1[1]*" \end{cases} \quad (1)$$

To sum up, `skey()` maps any node identifier to the storage key obtained by replacing the longest rightmost bit sequence of identical value by one single bit of same value.

A. SPT Construction and Maintenance

1) Insertion of New Summaries: To insert a new record within SPT, one performs three operations: (i) the lookup of the leaf node in charge of this summary; (ii) the determination of the DHT node where to store that leaf node thanks to function $skey()$; (iii) the storage of the new record.

When a leaf node reaches its maximum capacity, it splits into two leaf nodes. The left branch is labelled "0", while the right one is labelled "1". Each child node is assigned a subset of the content of the leaf node. Thanks to the specifics of our $skey()$ mapping function, the child node that has the same

rightmost bit as its parent is mapped to the same storage key as its parent. Hence, after a node splits, apart for the case of root node, only a subset of content is migrated to a diffrent DHT storage node.

2) Removals of Indexed Documents: To remove an existing record from SPT, the system proceeds the same way as for insertion. Firstly, it locates the leaf node in charge of that summary; then determines the storage key corresponding to that leaf node; finally, it removes the concerned record from stored bucket. After a removal, if the sum of records assigned to a node and its sibling falls under the maximal capacity of a leaf node, the contents of these two siblings must be merged.

Algorithm 1 *mergeIfNeeded(node)*: merge a node and its sibling if their total size is less than the maximum capacity of a leaf node

Require: $node$
1: **if** (node.content.size() $< (B \div 2)$ **then**
2: sibling $\leftarrow$ node.label; len $\leftarrow$ node.label.length();
3: **if** (node[node.length()-1] == 0) **then**
4: sibling[sibling.length()-1] $\leftarrow$ 1;
5: **else**
6: sibling[sibling.length()-1] $\leftarrow$ 0;
7: **end if**
8: ssKey $\leftarrow$ skey(sibling); snode $\leftarrow$ DHT-get(ssKey);
9: **if** ((snode.label.length(() $==$ node.label.length()) && (snode.content.size() + node.content.size() $< B$)) **then**
10: sum $\leftarrow$ node.content $\cup$ snode.content;
11: **if** (node.label[len-1] == node.label[len-2]) **then**
12: pn $\leftarrow$ node; dnode $\leftarrow$ snode;
13: **else**
14: pn $\leftarrow$ snode; dnode $\leftarrow$ node
15: **end if**
16: pn.label$\leftarrow$ pn.label.substring(0,pn.label.length()-1);
17: pn.content $\leftarrow$ sum; psk $\leftarrow$ skey(pn.label);
18: DHT-put(psk, pn);
19: dnode.content $\leftarrow \emptyset$; dnode.status $\leftarrow$ EXTERNAL;
20: dskey $\leftarrow$ skey(dnode.label); DHT-put(dskey, dnode);
21: **end if**
22: **end if**

SPT relies on function `mergeIfNeeded()` (see Alorithm 1) to merge sibling nodes if such an action is required. For that, first, the function checks if the number of remaining records is less than $(\frac{B}{2})$ where B is the maximum capacity of leaf node (line 1). If this condition is satisfied, the function computes the identifier of the sibling node then retrieves its corresponding data from the DHT store (lines 2 – 8). Once data is retrieved, the merging process continues if two conditions are satisfied: (i) the sibling did not yet split and (ii) the size of the union set of content is less than B (line 9). If a merging is required, as for split, the content of the sibling with its rightmost bit different from the rightmost bit of the parent is added to the contents of the other. Then both siblings are updated, the one with the same rightmost bit as the parent is updated to become the parent while the other is updated to become an external node (lines 10 – 21)

B. SPT-Lookup Primitive

This primitive returns the identifier of the SPT node in charge of the key passed at the call time. Algorithm 2 sketches how this primitive proceeds.

Algorithm 2 SPT-Lookup(sid, key) : looks up the node in charge of `key` located within the sub-tree identified by sid

Require: sid // The identifier of the subtree where to look up
Require: key // The summary to locate
Ensure: nid // The identifier of the node in charge of key
1: prefix $\leftarrow$ sid; inc $\leftarrow$ ""; notYetFound $\leftarrow$ true;
2: **while** (notYetFound) **do**
3: prevPrefix $\leftarrow$ prefix; prefix $\leftarrow$ prevPrefix $\odot$ inc;
4: rest $\leftarrow$ key.substring(prefix.length()-1, key.length());
5: rlen = rest.length(); dhtKey $\leftarrow$ skey(prefix);
6: node $\leftarrow$ `DHT-get(dhtKey)`;
7: **if** (node.status == EXTERNAL) **then**
8: prefix $\leftarrow$ prevPrefix; mid $\leftarrow$ inc.length() $\div$ 2;
9: inc $\leftarrow$ inc.substring(0, mid);
10: **else**
11: **if** ((node.status == INTERNAL) $\|$ ((isPrefix(node.label, key) == false))) **then**
12: inc $\leftarrow \mathcal{SPW}one$(rest, rlen);
13: **else**
14: nid $\leftarrow$ node.label; notYetFound $\leftarrow$ false;
15: **end if**
16: **end if**
17: **end while**
18: **return** nid;

`SPT-lookup` checks successively the subtree root node and its descendants leaf nodes identified by labels obtained by concatenating to "/" the prefixes of `key` with the rightmost bit equal to 1. These prefixes are considered in their length order and are constructed incremently thanks to the function $\mathcal{SPW}one$ which determines, at each time, the increment to add to the previous prefix to obtain the next satisfying prefix. In case `SPT-lookup` jumps to an external node after adding some increment, it steps back by reducing the length of the last added increment by half. Once the leaf node in charge of `key` is reached (that is a leaf node whose the identifying label is a "/" concatenated to a prefix of `key`), this label is returned and the `SPT-lookup` primitive terminates.

Compared to the traditional linear and binary lookup methods, `SPT-lookup` is somewhat hybrid. It worths to note that the number of `DHT-get()` performed in order to locate the SPT node in charge of `key` is less or equal to $n + 2$, where n is the number of bits 1 within `ikey`.

Note that `SPT-lookup` relies on two basic functions: $isPrefix()$ and $\mathcal{SPW}one$. Function $isPrefix()$ returns true if the first string is a prefix of the second one, while $\mathcal{SPW}one(bs, l)$ returns the shortest prefix of bs with at most l bits and its rightmost bit equal to 1 if such a prefix exists;

otherwise 0 (see Equation 2). In this equation, z is a sequence bits 0 while [0—1]* is any sequence of bits.

$$\mathcal{SPWone}(s,l) = \begin{cases} z1, & if\ s = z1[0|1]*\ and\ len(z) < l \\ 0, & if\ s = z[0|1]*\ and\ len(z) \geq l \\ 1, & otherwise \end{cases} \quad (2)$$

V. SPT Superset Search

Let qs be some keyword-set summary. The superset search primitive (Algorithm 3) aims to determine the set $\mathcal{R}$ of indexed summaries that are descendants of qs (i.e., $\forall r \in \mathcal{R}, r \hookrightarrow qs$).

Algorithm 3 SPT-supersetSearch(qs): returns the set of indexed summaries that contain qs

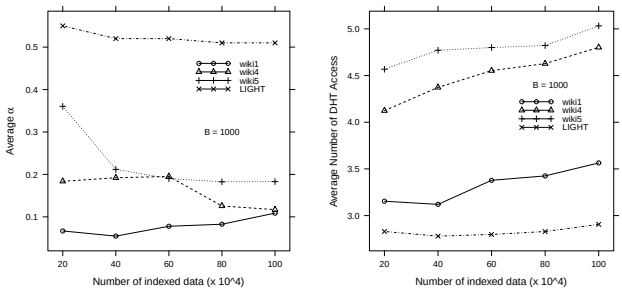
Require: qs
Ensure: $\mathcal{R}$
1: $\mathcal{R} \leftarrow \emptyset$; rp $\leftarrow$ "/"; bs.push(rp);
2: **while** (bs $\neq \emptyset$) **do**
3: bid $\leftarrow$ bs.pop(); blen $\leftarrow$ bid.length();
4: **if** (bid[blen -1] == 1) **then**
5: nid $\leftarrow$ bid;
6: **else**
7: ks $\leftarrow$ qs.substring(blen -1, qs.length);
8: ssk $\leftarrow$ bid $\odot$ ks; nid $\leftarrow$ SPT-lookup(bid, ssk);
9: **end if**
10: dhtKey $\leftarrow$ skey(nid); node $\leftarrow$ DHT-get(dhtKey);
11: sset $\leftarrow$ node.retrieveSupset(qs); $\mathcal{R} \leftarrow \mathcal{R} \cup sset$;
12: ibs $\leftarrow$ getBranches(node.label, bid);
13: **while** (ibs $\neq \emptyset$) **do**
14: ib $\leftarrow$ ibs.pop(); bs.push(ib);
15: **end while**
16: **end while**
17: **return** $\mathcal{R}$;

The search protocol is mainly implemented by the loop from Line 2 through Line 16. At each round the search protocol performs four actions. Firstly, it set `bid`, the identifier of the subtree where to search for this round. This is done simply by taking the node identifier on top of the stack `bs` (line 3). Secondly, the protocol determines `dhtKey`, the DHT storage key of either the rightmost or the leftmost descendant of node identified by `bid` that can store supersets of qs (lines 4 – 10). Once `dhtKey` is computed, the third action is to access the DHT and to retrieve the bucket stored within the DHT with `dhtKey`. Upon reception of this bucket, summaries that are superset of qs are added to the set $\mathcal{R}$ of responses. The final action of each round is to determine the set of new pertinent subtrees that need to be explored (line 12) and adds its elements to the current `bs` (lines 13 – 15). Note that, given a label `p`, branches of `p` are the set of subtrees identifiers obtained by changing the rightmost bit of each prefix of `p`. Function `getBranches(path, ancestor)` returns the set of identifiers of interesting branches of `path`, descendants of `ancestor`.

Once `bs` becomes empty, $\mathcal{R}$ contains all indexed summaries that are supersets of qs (line 17).

TABLE I
DATASETS DESCRIPTIONS

Dataset name	Number of words in a summary	number of summary
wiki1	$1 \leq$ nb words < 10	481380
wiki4	$40 \leq$ nb words < 60	618164
wiki5	$60 \leq$ nb words < 80	336373



(a) (b)

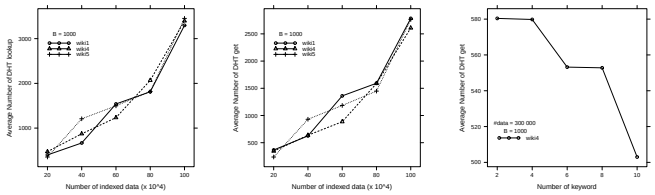
Fig. 1. Split cost (a) and lookup performance (b) as a function of the index size

VI. Performance Evaluation

This section presents the performances of a prototype implementation of SPT. We implemented SPT in java and simulated a DHT with an array of storage nodes, each capable to store B records. For this evaluation, we considered a real dataset composed of 4,636,000 abstracts of Wikipedia[1], which was subsequently treated and subdivided into smaller derived datasets. Table I shows the characteristics of the 3 derived datasets used for the evaluation. For all the experiments, each document is summarized by a Bloom filter of 1024 bits, using 5 hash functions. Due to space limitation we focus on three metrics: the split cost, the performances of SPT for the lookup and of the superset search operations. We compare the split cost and the lookup performance for the SPT proposal with the ones of LIGHT [8]. For this comparison, we convert each description into a float number. It worths to note that such a conversion can have an impact on this comparison.

a) SPT Maintenance Cost: We measure the ratio, α, of records moved to a remote peer during a leaf split. For that, we continuously insert data into the index and log the average value of α at each split. During this experiment B is fixed and is equal to 1000. Figure 1a plots the ratio of data moved during our simulation. For SPT, in average, less than 20 percent of data are migrated to a remote node after a leaf split; also, the higher the number of keywords in each description, the higher the average ratio of data moved to a remote DHT node on each split. Compared to LIGHT, SPT offers better performances thought one needs to evaluate the impact of data conversion on the LIGHT performances.

[1] http://data.dws.informatik.uni-mannheim.de/dbpedia/2014/en/long_abstracts_en.ttl.bz2.



(a) (b) (c)

Fig. 2. Performance of the superset search operation as a function of the index size (a and b) and of the number of kywords in the request (c)

b) SPT Lookup Performance: To evaluate the lookup primitive, we run three series of experiments, one for each dataset. For each experiment, we first insert a predefined number of records from one dataset within the index, then run 1000 lookups operations for documents peaked from the same dataset. We record the number of DHT-accesses for each lookup operation.

Figure 1b reports the average number of DHT accesses per lookup for each experiment as a function of the index size and the dataset. The number of DHT accesses required to complete a SPT-lookup varies between 2 and 7. Also, the number of DHT access increases, though very smoothly, as the size of the data increases. This figures shows finally that LIGHT performs better than SPT.

c) SPT Superset Search Performance: We run experiments with different index sizes while maintaining B equal to 1000. For each experiment, after the insertion of the suitable number of records, we perform a number of superset searches and measure the average number of lookup and of get operations performed on the underlining DHT in order to retrieve all indexed summaries that are descendants of the query summary. It worths to note that the number of DHT get performed on behalf of a search request corresponds to the number of SPT leaf nodes whose labels are prefixes of supersets of this request summary and which must be accessed in order to retrieve the set of supersets that satisfy the search request. Note also that one SPT-lookup (cf. Algorithm 3, lines 7 – 8) is required to determine each SPT leaf node responsible of a prefix of a superset of a request summary. From figures 2a and 2b, we observe that, for each experiment, the average number of DHT-lookup is less than twice the average number of DHT-get. This suggests that our search protocol is particularly efficient in locating SPT leaf nodes that potentially store supersets of the query summary.

We also evaluate the performance of the search according to the number of keywords contained in a query. Figure 2c shows that the number of DHT-get decreases when the number of keywords in a search request increases.

VII. Conclusion

We presented SPT, a trie index data structure that can help build a scalable, fault-tolerant and robust over DHT index for keyword-based search. Our solution uses an hybrid lookup procedure that suits the specific of sparse indexing keys. Our extensive experimental evaluation with Wikipedia dataset comprising millions of documents demonstrates the efficency of our solution. Few data are moved during split operation and the SPT lookup operations are efficient reducing the search cost to the minimum necessary. We are currently working on improving our proposal to compress Bloom filters in order to reduce the storage cost of DHT nodes.

References

[1] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '03. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 21–40. [Online]. Available: http://dl.acm.org/citation.cfm?id=1515915.1515918

[2] M.-F. Wang, D.-F. Zhang, X.-M. Tian, xia-an Bi, and B. Zeng, "Multi-keyword search over p2p based on counting bloom filter," in *2nd International Conference on Networking and Information Technology (IPCSIT)*, 2009.

[3] H. Chen, H. Jin, L. Chen, Y. Liu, and L. M. Ni, "Optimizing bloom filter settings in peer-to-peer multikeyword searching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 4, pp. 692–706, 2012.

[4] H. Chen, H. Jin, J. Wang, L. Chen, Y. Liu, and L. M. Ni, "Efficient multi-keyword search over p2p web," in *17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 989–998. [Online]. Available: http://doi.acm.org/10.1145/1367497.1367631

[5] L. Liu, K. D. Ryu, and K.-W. Lee, "Supporting efficient keyword-based file search in peer-to-peer file sharing systems," in *IEEE Global Telecommunications Conference (GLOBECOM '04)*, vol. 2, 2004, pp. 1259–1265.

[6] Y.-J. Joung, C.-T. Fang, and L.-W. Yang, "Keyword search in dht-based peer-to-peer networks," in *25th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 339–348. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2005.44

[7] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, "Brief announcement: Prefix hash tree," in *23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, 2004.

[8] Y. Tang, S. Zhou, and J. Xu, "Light: A query-efficient yet low-maintenance indexing scheme over dhts," *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, pp. 59–75, 2010. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2009.47

[9] N. Hidalgo, L. Arantes, P. Sens, and X. Bonnaire, "A tabu based cache to improve latency and load balancing on prefix trees," in *17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Tainan, Taiwan, 2011, pp. 557–564. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2011.18

[10] R. Cortés, X. Bonnaire, O. Marin, L. Arantes, and P. Sens, "Geotrie: A scalable architecture for location-temporal range queries over massive geotagged data sets," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, Oct 2016, pp. 10–17.

[11] A. S. Tigelaar, D. Hiemstra, and D. Trieschnigg, "Peer-to-peer information retrieval: An overview," *ACM Trans. Inf. Syst.*, vol. 30, no. 2, pp. 9:1–9:34, 2012. [Online]. Available: http://doi.acm.org/10.1145/2180868.2180871

[12] X. Jin, W.-P. K. Yiu, and S.-H. G. Chan, "Supporting multiple-keyword search in a hybrid structured peer-to-peer network," in *IEEE International Conference on Communications (ICC)*. IEEE Computer Society, June 2006, pp. 42–47. [Online]. Available: http://dblp.uni-trier.de/db/conf/icc/icc2006.html\#JinYC06

[13] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over dht," in *Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[14] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000. [Online]. Available: http://dx.doi.org/10.1109/90.851975